Programmazione in Java

a cura di Francesco Galgani
(www.galgani.it)

23 aprile 2006

Indice

Capitolo 1

Introduzione

... è quasi sempre scorretto iniziare la decomposizione in moduli di un sistema sulla base di un diagramma di flusso. Noi proponiamo invece che uno inizi con una lista di difficili decisioni di progettazione o di decisioni di progettazione che probabilmente cambieranno. Ogni modulo va poi progettato per nascondere una tale decisione agli altri.

Vorrei raccomandare gli studenti di porre maggiore attenzione alle idee fondamentali piuttosto che all'ultima tecnologia. La tecnologia sarà vecchia prima che loro saranno laureati. Le idee fondamentali non invecchieranno mai. Comunque, quello che mi preoccupa riguardo a quanto ho appena detto, è che alcune persone penserebbero alle macchine di Turing e al teorema di Goedel come fondamentali. Io penso che queste cose sono fondamentali ma anche pressoché irrilevanti. Penso che ci sono dei fondamentali principi di progettazione, per esempio i principi della programmazione strutturata, le buone idee nella programmazione "Object Oriented", ecc..

David Parnas

testo originale in inglese: http://en.wikipedia.org/wiki/David_Parnas

1.1 Testi consigliati

Letture di riferimento, gratuite e liberamente consultabili on-line, per iniziare lo studio di Java:

- **Java.sun.com, the source for Java developers**
  - The Java Tutorial, a practical guide for programmers
    http://java.sun.com/docs/books/tutorial/index.html
  - JavaTM 2 Platform Standard Ed. 5.0 - API Specification
    http://java.sun.com/j2se/1.5.0/docs/api/
- **Mokabyte.it, rivista italiana on-line dedicata a Java**
  - Corso Introduttivo su Java
    Indice (non presente su Mokabyte):
    http://www.mokabyte.it/2002/04/javabase-3.htm
    http://www.mokabyte.it/2002/06/javabase-4.htm
    http://www.mokabyte.it/2002/07/javabase-5.htm
    http://www.mokabyte.it/2002/09/javabase-6.htm
    http://www.mokabyte.it/2002/10/javabase-7.htm
    http://www.mokabyte.it/2002/11/javabase-8.htm
    http://www.mokabyte.it/2002/12/javabase-9.htm
    http://www.mokabyte.it/2003/02/corsojava-10.htm

http://www.mokabyte.it/2003/03/corsojava-11.htm
http://www.mokabyte.it/2003/04/corsojava-12.htm
http://www.mokabyte.it/2003/05/corsojava-13.htm
http://www.mokabyte.it/2003/06/corsojava-14.htm
http://www.mokabyte.it/2003/09/corsojava-15.htm
http://www.mokabyte.it/2003/10/corsojava-16.htm
http://www.mokabyte.it/2003/11/corsojava-17.htm

- **Risorse didattiche per l'ingegneria del software, di Claudio De Sio Cesari**
  - **Object Oriented && Java 5**
    manuale completo in italiano su Java versione 5
    PDF di 600 pagine con esercizi, 5 sezioni, 19 moduli, 8 appendici, 85 unità
    http://www.claudiodesio.com/download/oo_&&_java_5.zip

- **Latemar.science.unitn.it, corsi tenuti da Marco Ronchetti**
  - Corso di Programmazione 2
    http://latemar.science.unitn.it/marco/Didattica/aa_2005_2006/P2/index.html

1.2 Perché la programmazione ad oggetti?

tratto da *"Dai fondamenti agli oggetti"*[1]:

« Tutti i linguaggi di programmazione forniscono meccanismi di astrazione. Un linguaggio assembler, che costituisce il livello di astrazione più elementare, offre semplicemente una rappresentazione simbolica delle istruzioni del processore. I linguaggi di programmazione imperativi, come Pascal, C e Fortran, forniscono invece astrazioni più o meno raffinate della macchina sottostante; in particolare, i linguaggi imperativi "ad alto livello" mettono a disposizione strutture di controllo che sono astrazioni delle istruzioni di controllo disponibili in assembler, tipi primitivi che sono astrazioni dei tipi disponibili in assembler e meccanismi per raffinare tali astrazioni (definizioni di nuovi tipi e nuove istruzioni).

Sebbene queste astrazioni costituiscano un notevole passo avanti rispetto a quelle fornite dal linguaggio assembler, restano ancora saldamente legate alla struttura del calcolatore. Dovendo sviluppare un programma per risolvere un problema, il programmatore è obbligato a progettare e a realizzare il programma in termini della struttura del calcolatore sottostante anziché nei termini della struttura del problema. In sostanza, il linguaggio fornisce un mondo *o* dominio della soluzione *(il mondo del calcolatore) in cui il programmatore deve simulare gli elementi che compaiono nel dominio del problema introducendo opportune traduzioni; la soluzione del problema è quindi realizzata manipolando tali rappresentazioni tramite le istruzioni del linguaggio. Non a caso, uno dei passi fondamentali nella progettazione di un programma imperativo consiste nell'individuare la rappresentazione dei* dati *rilevanti per la soluzione del problema. La difficoltà di realizzare la corrispondenza fra dominio del problema e dominio della soluzione dà luogo a soluzioni innaturali difficilmente comprensibili in termini di dominio del problema; questo ha un grosso impatto sul processo di produzione e di manutenzione dei programmi.*

I linguaggi di programmazione ad oggetti, come ad esempio Java, forniscono invece astrazioni che consentono di rappresentare direttamente nel dominio della soluzione gli elementi che compaiono nel dominio del problema. In questo modo, la corrispondenza fra i due domini può essere espressa chiaramente: a ogni agente*, cioè a ogni* oggetto *che compare nel dominio del problema, corrisponde un analogo* oggetto software *nel dominio della soluzione; il programmatore può quindi cercare di risolvere il problema simulando nel dominio della soluzione il modo in cui risolverebbe il problema nel mondo reale.*

[...] »

[1] "Dai fondamenti agli oggetti", di G. Pighizzini e M. Ferrari, ed. Pearson Education Italia, http://hpe.pearsoned.it/

1.2.1 Un diverso paradigma di programmazione

Almeno a livello intuitivo, è quindi possibile stabilire una basilare distinzione:

- Nella programmazione "tradizionale", i problemi vengono affrontati rivolgendo l'attenzione alle operazioni da svolgere: partendo dal "main", si arriva ad una decomposizione funzionale del problema.
- Nella programmazione "ad oggetti", si vuole innanzitutto stabilire quali sono le entità che interagiranno tra loro, ricreando un modello del mondo reale. Tali entità sono gli *oggetti.*

1.2.2 Le classi e gli oggetti

Una classe è intuitivamente la descrizione astratta di una categoria di oggetti, dotati di *stato (proprietà)* e di *comportamenti (metodi)*. Una classe rappresenta quindi una categoria di oggetti, ad esempio esprime l'idea di "gatto", definendone lo stato (sesso, età, razza, peso, tipo di pelo, lunghezza della coda, ecc.) e i comportamenti (fare le fusa, miagolare, correre, mangiare, dormire, saltare, ecc.), mentre gli oggetti sono tutti i singoli gatti istanziati dalla classe.

Gli obiettivi fondamentali delle classi, come indicato nella *Introduzione alla OOP (Object Oriented Programming)* di SuperEva[2], sono:

1. definire astrazioni, cioè approssimazioni di un oggetti reali che hanno un ruolo nel problema da risolvere, limitatamente a tutti e soli gli aspetti interessanti;
2. favorire la modularità, cioè suddividere le operazioni del programma in blocchi;
3. implementare l'*information hiding* (criterio di Parnas per il quale ogni classe "nasconde" alcune informazioni che sono utili solamente allo svolgimento dei suoi compiti).

Ogni classe, oltre a nascondere la sua implementazione, espone un'interfaccia, cioè l'insieme di dati *(proprietà)* e di comportamenti *(metodi)* che sono visibili nel resto del programma.

1.2.3 Livelli di astrazione

Maggiore è la complessità di un problema e maggiore sarà il livello di astrazione necessario, che si concretizzerà nella necessità di definire tipi di dati personalizzati e complessi.

E' indispesabile una "composizione di concetti": partendo dalle cose più semplici, si arriva a quelle più complesse per livelli di astrazione, tralasciando di volta in volta i dettagli. E' possibile dominare la complessità con una organizzazione gerarchica, cioè decomponendo la realtà in blocchi che nascondono la loro intrinseca complessità e creando oggetti via via sempre più complessi.

Java offre una miriade di dati predefiniti, ad esempio per creare le interfacce grafiche. Tale approccio permette di ragionare ad alto livello, senza alcuna preoccupazione per cosa faccia la macchina a livello più basso. A differenza del C++, dove viene lasciata ampia libertà nella scelta di uno stile di programmazione imperativo od orientato agli oggetti, tutto il codice Java è invece diviso in classi.

1.2.4 Dalla struttura alla classe

Formalmente una *classe* costituisce un *modello* che descrive l'informazione e il comportamento associati con le *istanze* della classe stessa: istanziare una classe significa creare un *oggetto* basato sul modello della classe e pertanto simile alle altre istanze della medesima classe. L'informazione associata ad una classe o ad un oggetto è immagazzinata nelle variabili, mentre il comportamento

[2] http://guide.supereva.com/c_/interventi/2000/08/10150.shtml

è implementato con i *metodi*, che sono simili alle funzioni (o procedure) nei linguaggi procedurali come il C.

A partire dal concetto di *struct* del C, infatti, è possibile arrivare a quello di classe completando la struct con funzioni che vanno ad agire sui dati in essa contenuti: le funzioni non sono quindi più slegate dai dati su cui agiscono, ma insieme ad essi formano un'unica entità. I metodi, cioè le funzioni integrate nella classe, possono essere pubblici o privati: quelli pubblici permettono alla classe di rapportarsi con l'esterno, quelli privati fanno invece strettamente parte della sua implementazione e non devono essere visibili al di fuori della classe stessa.

Come ulteriore esemplificazione del rapporto tra classi e oggetti, si consideri una classe che rappresenti un rettangolo. Tale classe conterrà variabili per indicarne l'origine, la larghezza e l'altezza (queste sono informazioni) ed eventualmente un metodo per calcolarne e restituirne l'area (questo è un comportamento). Un'istanza della classe rettangolo conterrà le informazioni di uno specifico rettangolo, come le dimensioni di un pavimento o di questa pagina.

1.2.5 Separazione di interfaccia e implementazione

La separazione di interfaccia e implementazione è uno dei concetti più importanti nella progettazione di software. Tenendo presente sia il fatto che le componenti dei software sono spesso sviluppate in parallelo da diversi programmatori, isolati l'uno dall'altro, sia la crescente importanza del riutilizzo di parti di codice all'interno di progetti diversi, ne consegue la necessità di avere delle interconnessioni minime e ben note tra le varie porzioni di un sistema software. David Parnas, uno dei pionieri dell'Ingegneria del Software, ha espresso queste idee in un paio di regole che portano il suo nome (princìpi di Parnas[3]):

- Lo sviluppatore di un componente software deve dare all'utente le informazioni necessarie per fare un uso effettivo dei servizi forniti dal componente, e non dovrebbe fornirgli nessun'altra informazione.
- Lo sviluppatore di un componente software deve essere messo a conoscenza di tutte le informazioni necessarie per portare a termine le responsabilità richieste e assegnate al componente, e non dovrebbe ricevere ulteriori informazioni.

1.3 Un primo sguardo al linguaggio

1.3.1 La "Java Virtual Machine" e i punti di forza di Java

tratto da *"Caratteristiche di Java", di Claudio De Sio Cesari*[4]:

« Java ha alcune importanti caratteristiche che permetteranno a chiunque di apprezzarne i vantaggi.

- ***Sintassi**: è simile a quella del C e del C++, e questo non può far altro che facilitare la migrazione dei programmatori da due tra i più importanti ed utilizzati linguaggi esistenti. Chi non ha familiarità con questo tipo di sintassi, può inizialmente sentirsi disorientato e confuso, ma ne apprezzerà presto l'eleganza e la praticità.*
- ***Gratuito**: per scrivere applicazioni commerciali non bisogna pagare licenze a nessuno. Sun ha sviluppato questo prodotto e lo ha migliorato usufruendo anche dell'aiuto della comunità "open-source".*

[3] testo originale in inglese: http://homepages.north.londonmet.ac.uk/~mulhollm/im52P/week1-2lec.html
[4] http://www.wmlscript.it/java/corso_02.asp
riportato anche su: http://www.webmasterpoint.org/java2/02.asp
homepage dell'autore: http://www.claudiodesio.com/

- ***Robustezza***: *essa è derivante soprattutto da una gestione delle eccezioni chiara e funzionale, e da un meccanismo automatico della gestione della memoria (Garbage Collection) che esonera il programmatore dall'obbligo di dover deallocare memoria quando ce n'è bisogno, punto tra i più delicati nella programmazione. Inoltre il compilatore Java è molto "severo". Il programmatore è infatti costretto a risolvere tutte le situazioni "poco chiare", garantendo al programma maggiori chance di corretto funzionamento.*

- ***Libreria e standardizzazione***: *Java possiede un'enorme libreria di classi standard che forniscono al programmatore la possibilità di operare su funzioni comuni di sistema come la gestione delle finestre, dei collegamenti in rete e dell'input/output. Il pregio fondamentale di queste classi sta nel fatto che rappresentano un'astrazione indipendente dalla piattaforma, per un'ampia gamma di interfacce di sistema utilizzate comunemente. Inoltre, grazie alle specifiche di Sun, non esisteranno per lo sviluppatore problemi di standardizzazione, come compilatori che compilano in modo differente.*

- ***Indipendenza dall'architettura***: *grazie al concetto di macchina virtuale ogni applicazione, una volta compilata, potrà essere eseguita su di una qualsiasi piattaforma (per esempio un PC con sistema operativo Windows o una workstation Unix). Questa è sicuramente la caratteristica più importante di Java. Infatti, nel caso in cui si debba implementare un programma destinato a diverse piattaforme, non ci sarà la necessità di doverlo convertire radicalmente da piattaforma a piattaforma. E' evidente quindi che la diffusione di Internet ha favorito e favorirà sempre di più la diffusione di Java.*

- ***Java Virtual Machine***: *ciò che di fatto rende possibile l'indipendenza dalla piattaforma è la Java Virtual Machine (da ora in poi JVM), un software che svolge un ruolo da interprete (ma non solo) per le applicazioni Java. Più precisamente, dopo aver scritto il nostro programma Java, prima bisogna compilarlo. Otterremo così, non direttamente un file eseguibile (ovvero la traduzione in linguaggio macchina del file sorgente che abbiamo scritto in Java), ma un file che contiene la traduzione del nostro listato in un linguaggio molto vicino al linguaggio macchina detto "bytecode". Una volta ottenuto questo file dobbiamo interpretarlo. A questo punto la JVM interpreterà il bytecode ed il nostro programma andrà finalmente in esecuzione. Quindi, se una piattaforma qualsiasi possiede una Java Virtual Machine, ciò sarà sufficiente per renderla potenziale esecutrice di bytecode. Infatti, da quando ha visto la luce Java, i Web Browser più diffusi implementano al loro interno una versione della JVM, capace di mandare in esecuzione le applet Java. Ecco quindi svelato il segreto dell'indipendenza della piattaforma: se una macchina possiede una JVM, può eseguire codice Java.*
  N.B. : Un browser mette a disposizione solamente una JVM per le applet non per le applicazioni standard. Si parla di "macchina virtuale" perché in pratica questo software è stato implementato per simulare un hardware. Si potrebbe affermare che il linguaggio macchina sta ad un computer come il bytecode sta ad una Java Virtual Machine. Oltre che permettere l'indipendenza dalla piattaforma, la JVM permette a Java di essere un linguaggio multi-threaded (caratteristica di solito dei sistemi operativi), ovvero capace di mandare in esecuzione più processi in maniera parallela. Inoltre, garantisce dei meccanismi di sicurezza molto potenti, la "supervisione" del codice da parte del Garbage Collector, validi aiuti per gestire codice al runtime e tanto altro...

- ***Orientato agli oggetti***: *Java ci fornisce infatti degli strumenti che praticamente ci "obbligano" a programmare ad oggetti. I paradigmi fondamentali della programmazione ad oggetti (ereditarietà, incapsulamento, polimorfismo) sono più facilmente apprezzabili e comprensibili. Java è più chiaro e schematico che qualsiasi altro linguaggio orientato agli oggetti. Sicuramente, chi impara Java, potrà in un secondo momento accedere in modo più naturale alla conoscenza di altri linguaggi orientati agli oggetti, giacché, avrà di certo una mentalità più "orientata agli oggetti".*

- ***Semplice***: *riguardo quest'argomento, in realtà, bisogna fare una precisazione. Java è un linguaggio molto complesso considerandone la potenza e tenendo presente che ci obbliga ad imparare la programmazione ad oggetti. Ma, in compenso, si possono ottenere risultati insperati in un tempo relativamente breve. Apprezzeremo sicuramente le semplificazioni che ci offre Java. Abbiamo per esempio già accennato al fatto che non esiste l'aritmetica dei puntatori grazie all'implementazione della Garbage Collection. Provocatoriamente Bill Joy, vice-presidente della Sun Microsystems negli*

anni in cui nacque il linguaggio, propose come nome (alternativo a Java) "C++–". Questo per sottolineare con ironia che il nuovo linguaggio voleva essere un nuovo C++, ma senza le sue caratteristiche peggiori (o se vogliamo, senza le caratteristiche più difficili da utilizzare e quindi pericolose).

- ***Sicurezza**: ovviamente, avendo la possibilità di scrivere applicazioni interattive in rete, Java possiede anche delle caratteristiche di sicurezza molto efficienti. Come c'insegna la vita quotidiana, nulla è certo al 100%. Esistono una serie di problemi riguardanti la sicurezza di Java che ricercatori dell'Università di Princeton hanno in passato scoperto e reso pubblici su Internet. Ma di una cosa però possiamo essere certi: il grande impegno che Sun dedica alla risoluzione di questi problemi. Intanto, oggi come oggi, Java è semplicemente il linguaggio "più sicuro" in circolazione.*
- ***Risorse di sistema richieste**: e questo è il punto debole più evidente di Java. Infatti, non esistendo l'aritmetica dei puntatori, la gestione della memoria è delegata alla Garbage Collection della JVM. Questa garantisce il corretto funzionamento dell'applicazione (ovvero non dealloca la memoria che è ancora utilizzabile), ma non favorisce certo l'efficienza. Inoltre i tipi primitivi di Java non si possono definire "leggeri". Per esempio i caratteri sono a 16 bit, le stringhe immutabili, e non esistono tipi senza segno (gli "unsigned").*

[...] »

1.3.2 Compilazione ed esecuzione

In sintesi, il codice sorgente di una classe deve essere contenuto in un file di testo che ha lo stesso nome della classe e estensione ".java", la compilazione produrrà un file ".class" (contenente bytecode) e, in esecuzione, verrà richiamato il nome della classe e non quello del file che la contiene.

Le istruzioni dettagliate per compilare ed eseguire un applicativo sono riportate sul sito della Sun, alla pagina:

http://java.sun.com/docs/books/tutorial/getStarted/cupojava/index.html

Sono comunque disponibili diversi IDE (Integrated Development Environment) per semplificare la progettazione e programmazione in Java, sia con licenza open-source, come Eclipse:

http://www.eclipse.org/

sia con licenze di tipo diverso, ma comunque con possibilità d'uso gratuito, come Borland JBuilder Foundation:

http://www.borland.com/downloads/download_jbuilder.html

1.3.3 Un programma "minimo"

Si consideri la seguente applicazione di esempio:

```
/**
 * La classe HelloWorldApp implementa un'applicazione che
 * semplicemente visualizza "Hello World!" sullo standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Visualizza la stringa.
    }
}
```

Il codice contiene un commento, la dichiarazione della classe `HelloWordApp` e del suo metodo `main`. Non sono presenti istanze della classe, cioè oggetti.

Il metodo `main` è simile alla funzione *main* del C e del C++. Quando l'interprete esegue un'applicazione, inizia richiamando il metodo `main` della classe, il quale, a sua volta, chiamerà tutti gli altri metodi necessari per eseguire il programma. Il `main` viene dichiarato nella forma:

```
public static void main (String[] args)
```

Sebbene sia possibile cambiare il nome dell'argomento, ad esempio "args" in "pluto", oppure invertire l'ordine di "public static" in "static public", è prassi comune usare la forma sopra riportata.

La dichiarazione del metodo `main` contiene i due modificatori `public` e `static` (vedi sezione 3.5). Il tipo restituito è `void`, ovvero non viene restituito alcun dato.

Il metodo `main` accetta un solo argomento: un array di elementi di tipo String, ognuno dei quali contiene uno degli eventuali parametri passati dalla riga di comando (`args.length` ne indica il numero). A differenza del C e del C++, non viene memorizzato il nome del programma, dato che è uguale al nome della classe in cui il metodo main è definito.

L'ultima istruzione costituisce un semplice metodo per visualizzare una stringa sullo schermo.

Capitolo 2

Gli elementi di base del linguaggio

Al di là degli aspetti teorici fin qui esposti per presentare la programmazione *object-oriented*, i seguenti concetti, perlopiù comuni a buona parte dei linguaggi, anche se con inevitabili differenze, sono da considerarsi come "prerequisiti" indispensabili *(per tali argomenti, si rimanda alle guide indicate nella sezione 1.1 "Testi consigliati")*:

- schema minimo di un'applicazione;
- tipi di dati primitivi e composti;
- promozioni e casting;
- parole riservate (keywords);
- dichiarazione di una variabile;
- visibilità delle variabili;
- assegnamento di un valore ad una variabile;
- strutture di controllo (costrutti condizionali e iterativi);
- organizzazione della memoria;
- concetto di puntatore e dereferenziazione;
- creazione di un oggetto (operatore `new`);
- dichiarazioni e chiamate di metodi;
- firme (signatures) di metodi, parametri formali, parametri attuali;
- passaggio di parametri per valore e per riferimento;
- passaggio di parametri dalla riga di comando;
- compilazione ed esecuzione da riga di comando;
- oggetti `String`;
- arrays;
- concetto di package;
- gestione delle eccezioni con istruzioni try / catch.

2.1 Classi e oggetti

Per raggiungere il pieno controllo sul linguaggio Java, è necessario aver ben capito la differenza concettuale tra *classe*, *istanza* e *reference*.

Una *classe* è come uno stampino che permette di produrre una molteplicità di *oggetti* simili tra loro. Una *classe* è presente in singola copia nella memoria del computer. Ogni volta che si ricorre all'operatore `new`, viene creata una nuova *istanza della classe*, cioè un *oggetto* conforme alle specifiche della *classe* stessa. La variabile a cui viene associato l'oggetto è soltanto un *reference*, cioè un riferimento all'area di memoria che contiene l'oggetto. Un oggetto può avere uno o più *reference*; nel caso non ne abbia nemmeno uno, allora tale oggetto è inutilizzabile e si limita ad occupare memoria.

Gli oggetti interagiscono tra loro scambiandosi messaggi attraverso l'invocazione di metodi e l'utilizzo di parametri: le variabili dei tipi dei dati primitivi (byte, short, char, int, float, double, bool) sono sempre passati per copia, mentre gli oggetti sono sempre passati per referenza (nel senso che vengono copiati gli identificatori degli oggetti e non gli oggetti stessi).

2.1.1 Gestione della memoria, finalizzatori e Garbage Collection

La memoria disponibile per i programmi è divisa in:

- Stack
  - *memoria allocata dai metodi per le variabili locali*
  - *le variabili non più utilizzate vengono automaticamente rimosse (le ultime inserite sono le prime ad essere eliminate)*
- Heap
  - *memoria allocata per gli oggetti*
  - *gli oggetti non più referenziati continuano ad occupare memoria finché non interviene il Garbage Collector oppure fino alla fine del programma*

Se la memoria disponibile scende al di sotto di una certa soglia, la pulizia delle aree allocate ad oggetti non più referenziati (quindi inutilizzabili) viene svolta automaticamente durante l'esecuzione del programma dal Garbage Collector (letteralmente "raccoglitore di rifiuti"). Java supporta il *multi-thread*, cioè è capace di gestire più operazioni insieme: questo permette al Garbage Collector di attivarsi senza causare blocchi temporanei del programma.

Prima di liberare le aree di memoria, il Garbage Collector invoca il metodo `finalize()` (se esiste) sull'oggetto da rimuovere. Implementare tale metodo è utile esclusivamente nel caso in cui sia necessario svolgere operazioni che non vengono compiute automaticamente dal Garbage Collector.

Per forzare la JVM ad eseguire il Garbage Collector è disponibile il comando `System.gc()`, eventualmente preceduto da `runFinalization()`.

2.1.2 Convenzioni di Naming

Esistono delle convenzioni universalmente accettate nel modo di attribuire i nomi che, benché siano irrilevanti ai fini della compilazione, sono invece di fondamentale importanza per la leggibilità del codice:

- i nomi di classi devono iniziare con una lettera maiuscola;
- i nomi di metodi e variabili iniziano con una lettera minuscola;

- i nomi composti usano la convenzione “Camel Case”: le parole vengono riportate in minuscolo, una di seguito all’altra senza caratteri di separazione, usando un carattere maiuscolo come lettera iniziale di ogni parola.

Ad esempio:

```
NomeClasse nomeMetodo() nomeVariabile
```

2.1.3 Incapsulamento

L’incapsulamento è il principio fondante del design object-oriented: il contenuto informativo di una classe deve rimanere nascosto all’utente, in modo tale che i metodi siano l’unica via per interagire con gli oggetti corrispondenti. Questo approccio ha due grandi vantaggi:

- l’incapsulamento permette al programmatore di disciplinare l’accesso agli attributi di una classe, in modo da evitare che ne venga fatto un uso sbagliato;
- l’incapsulamento permette all’utente di concentrarsi esclusivamente sull’interfaccia di programmazione, tralasciando ogni aspetto legato all’implementazione.

2.1.4 Costruttori

Il costruttore, che viene automaticamente invocato attraverso l’operatore `new`, è un particolare metodo che ha lo stesso nome della classe e che è privo di valore di ritorno. Il suo scopo è quello di inizializzare i principali attributi di un oggetto. Ad esempio:

```
public class Rettangolo {
    int base;
    int altezza;
    public Rettangolo(int x, int y) {
        base = x;
        altezza = y;
    }
    public void area () {
        System.out.println (base * altezza);
    }
    public static void main(String[] args) {
        Rettangolo rect = new Rettangolo(10, 20);
        rect.area();
    }
}
```

L’applicazione si comporterà in questa maniera:

1. viene invocato il metodo `main` della classe `Rettangolo`;
2. viene creata nell’heap un’istanza della classe `Rettangolo` con l’operatore `new`;
3. la posizione della locazione di memoria associata al nuovo oggetto viene memorizzata nell’identificatore di istanza `rect` (si tratta quindi di un puntatore memorizzato nello stack[1]);
4. nell’area di memoria del nuovo oggetto si trovano due variabili di tipo primitivo (`base` e `altezza`), che identificano lo stato dell’oggetto, mentre i metodi rimangono memorizzati nella classe (più oggetti dello stesso tipo, anche se hanno stati diversi, condividono gli stessi metodi della classe);
5. il metodo costruttore, che ha lo stesso nome della classe, riceve i parametri passati dal `main` (due `int`) e li utilizza per inizializzare le variabili della classe;

[1] *Java, a differenza del C e di altri linguaggi, semplifica il concetto di puntatore eliminando la possibilità di utilizzare l’“aritmetica dei puntatori”, impedendo di conseguenza al programmatore di accedere ad aree di memoria diverse da quelle referenziate attraverso la creazione di oggetti; tale approccio permette di evitare molti comuni errori di programmazione.*

6. il `main` richiama il metodo `area` dell'oggetto `rect`, il quale esegue il prodotto di `base` per `altezza`, lo converte automaticamente in un oggetto di tipo `String` e lo invia su standard output (cioè lo scrive sullo schermo);

7. fine del programma;

8. il sistema operativo dealloca le aree di memoria usate dal programma.

2.2 Javadoc

Per una descrizione dettagliata di Javadoc, vedi la documentazione fornita da Sun:

How to Write Doc Comments for the Javadoc Tool [2]

I commenti nella scrittura del codice sono utili per facilitarne la comprensione in fase di debug. Inoltre è buona regola inserire all'inizio di ogni applicazione alcune righe con la descrizione, l'autore, la versione e tutte le informazioni eventualmente utili.

I tre tipi possibili di commenti sono:

1. Commento su più righe:
```
/* bla
bla
bla */
```
2. Commento su una riga:
```
// bla
```
3. Commento usato da Javadoc:
```
/** bla
* bla
* bla */
```

Inserendo alcuni tags specifici, è possibile distribuire insieme al software la documentazione in formato HTML sulle classi e i metodi "public" e "protected". La funzione `javadoc` del JDK estrae tale informazioni e genera la documentazione.

- Tags per documentazione di classi:
  - @version
  - @author
- Tags per documentazione di metodi:
  - @para
  - @return
  - @exception

Esempio completo:

```
// http://xoomer.virgilio.it/gciabu/java/tut-java.htm
/**
* Esempi Commento della classe Commenti
* @author Gasparri Roberto
* @version 1.0
*/
public class Commenti {
    /**commento del metodo f()
    * @param parametri di ingresso
```

[2] http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

```
    * @return i valori che restituisce
    * @exception Eccezioni catturate/generate(trow)
    */
    public void f() { }
    public static void main(String args[])
}
/* Esempio di commento
su piu'linee */
```

Capitolo 3

Ereditarietà

Grazie all'ereditarietà è possibile definire una classe come figlia (sottoclasse o subclass) di una classe già esistente, allo scopo di realizzare estensioni *strutturali* della classe madre (superclasse o superclass), aggiungendo nuove variabili di istanza, od anche *comportamentali*, aggiungendo nuovi metodi o modificando quelli esistenti.

La sottoclasse, grazie alle direttiva `extends`, eredita quindi tutti i metodi e gli attributi della superclasse. L'ereditarietà permette di creare delle gerarchie di classi di profondità arbitraria, simili ad alberi genealogici, in cui il comportamento della classe in cima all'albero viene gradualmente specializzato nelle sottoclassi. Ogni classe può discendere da un'unica superclasse, mentre non c'è limite al numero di sottoclassi o alla profondità di derivazione. La classe `Object` è il capostipite di tutte le classi Java.

Nel seguente esempio, la classe `Rettangolo` viene estesa nella classe `Parallelepipedo`:

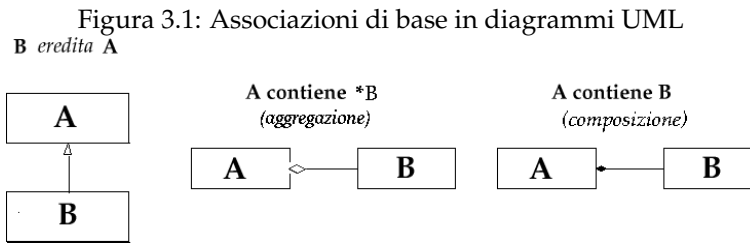
```
public class Parallelepipedo extends Rettangolo {
    int profondita;
    public Parallelepipedo(int x, int y, int z) {
        super(x,y);
        profondita = z;
    }
    public void area () {
        System.out.println(base * altezza * profondita);
    }
}
```

Si tenga presente che le variabili dichiarate `final` sono costanti, i metodi dichiarati `final` non possono essere sovrascritti e le classi dichiarate `final` non possono essere subclassate (vedi 3.5).

3.1 UML - Diagramma delle classi

Un diagramma UML:

- rappresenta le classi e gli oggetti che compongono il sistema, i relativi attributi ed operazioni;
- esiste una corrispondenza uno a uno tra gli elementi del diagramma e le dichiarazioni presenti nel codice.
- specifica, mediante le associazioni, i vincoli che legano tra loro le classi:
  - l'"ereditarietà" è rappresentata da una linea che parte dalla sottoclasse e si unisce alla superclasse con un triangolo (leggasi: "is a");
  - l'"aggregazione", cioè l'utilizzo di una classe all'interno di un'altra, viene indicata con un rombo bianco (leggasi: "has a");

Figura 3.1: Associazioni di base in diagrammi UML

– la "composizione", simile ad un'aggregazione ma con caratteristiche di esclusività, viene indicata con un rombo nero.

Le possibili associazioni di base sono schematizzate in figura 3.1.

Per una sintetica panoramica sui diagrammi UML, completa di esempi, vedi:
"Java e UML - Corrispondenza semantica tra Diagrammi di Classe e Codice Java"
di Andrea Gini [1].

3.2 Overloading

All'interno di una classe è possibile definire più volte un metodo, in modo da adeguarlo a contesti di utilizzo differenti. Due metodi con lo stesso nome possono coesistere in una classe se restituiscono lo stesso tipo e hanno firme differenti (firma = nome + tipo dei parametri).

Spesso una famiglia di metodi con lo stesso nome si limita a fornire diverse vie di accesso ad un'unica logica di base. Ad esempio, è possibile avere tre costruttori per la classe `Rettangolo`:

```
public Rettangolo(int x, int y) {
    base = x;
    altezza = y;
}
public Rettangolo (int x) {
    base = x;
    altezza = 10;
}
public Rettangolo () {
    base = 10;
    altezza = 10;
}
```

3.3 Overridding

Attraverso l'ereditarietà è possibile estendere il comportamento di una classe sia aggiungendo nuovi metodi, sia ridefinendo metodi già dichiarati nella superclasse. Questa seconda possibilità prende il nome di Overriding.

3.4 Identificatori this e super

L'identificatore `this` è un puntatore speciale alla classe che costituisce l'attuale contesto di programmazione. Grazie a `this` è possibile accedere a qualsiasi motodo o attributo della classe stessa attraverso espressioni del tipo: `this.metodo();` L'identificatore `this` può essere usato anche per richiamare un costruttore.

In maniera analoga, l'identificatore `super` è un puntatore speciale alla superclasse.

[1] http://www.lta.disco.unimib.it/didattica/IngSw/slide/Esercitazione-Java-UML.ppt

3.5 I modificatori

I modificatori sono parole riservate del linguaggio che permettono di impostare determinate caratteristiche di classi, metodi e attributi.

3.5.1 Modificatori di accesso

I modificatori di accesso permettono di impostare il livello di visibilità di classi, metodi e attributi. Ad ognuno di questi elementi è possibile assegnare uno dei seguenti livelli:

- nessun modificatore (package protection)
  l'elemento è visibile a tutte le classi che fanno parte dello stesso package
- public
  libero accesso: l'elemento è visibile ovunque
- protected
  accessibile dalle sottoclassi, indipendentemente dal package
- private
  accessibile solo all'interno della classe
- static
  accessibile anche senza creare istanze

I principi di base della programmazione ad oggetti, come l'incapsulamento, l'information hiding, la separazione di interfaccia e implementazione (concetti affini e determinanti), suggeriscono di rendere tutti gli attributi `private`, permettendone l'accesso indiretto attraverso il costruttore ed eventuali metodi `set`*`NomeVariabile`* o `get`*`NomeVariabile`*, e di dichiarare `private` anche quei metodi che sono richiamati esclusivamente all'interno della classe. E' consigliabile scegliere con cura quali metodi rendere pubblici e quali no, con la stessa accortezza che generalmente ha chi progetta un elettrodomestico nel non lasciare fili o ingranaggi scoperti.

3.5.2 Modificatore final

Il modificatore `final` assume un significatore diverso in base al contesto di utilizzo. Se abbinato ad un attributo, lo rende immutabile, con la conseguenza che è necessario assegnare una variabile final nello stesso momento della dichiarazione. Spesso `final` viene utilizzato in abbinameno a `static` per definire delle costanti:

```
public static final float pigreco = 3.14;
```

Se un reference ad un oggetto è `final`, risulta immutabile il reference, non l'oggetto. L'uso di `final` in abbinamento ad un metodo o ad una classe ha conseguenze per quanto riguarda l'ereditarietà: se abbinato ad un metodo, ne vieta l'overriding nelle sottoclassi; se associato ad una classe, proibisce la creazione di sottoclassi.

3.5.3 Modificatore abstract

Un metodo `abstract` è un metodo che non implementa un proprio blocco di codice, ad esempio:

```
public abstract void mioMetodo();
```

Questo metodo, ovviamente non richiamabile, può essere riscritto (override) in una sottoclasse e definito esclusivamente all'interno di una classe a sua volta `abstract`, la quale, benché abbia la caratteristica di non essere direttamente istanziabile, può essere implementata nelle sottoclassi.

Le classi astratte sono utili nella definizione di tipi di dati troppo generici per essere istanziati: la ramificazione in sottoclassi sarà utile per definire specifici tipi di dati a partire da quelli più generici (in questo caso, "astratti").

Capitolo 4

Interfacce e Polimorfismo

Se si considerano le conseguenze dell'ereditarietà, si scopre che ogni classe ha come tipo sia il proprio, sia quello di tutte le sue superclassi: tale osservazione sarà la base per introdurre il concetto di polimorfismo*. Una volta definiti quali servizi di base sono comuni a un insieme di classi (concetto di* interfaccia*) e abbinando questa caratteristica alle possibilità offerte dal polimorfismo, sarà possibile scrivere codice riutilizzabile in tante diverse situazioni.*

Per un eventuale approfondimento dei principi esposti in questo capitolo, vedi:

Programmazione Orientata agli Oggetti
Interfacce e Polimorfismo / Upcasting e downcasting [1]

4.1 Le interfacce

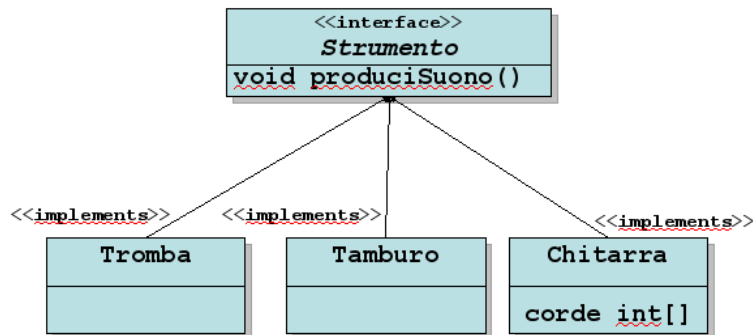
Oltre che per mezzo delle classi, è possibile definire nuovi tipi con il costrutto `Interface`:

- un'interfaccia permette di definire un tipo sulla base dei servizi offerti, ovvero dei metodi di cui dispone;
- non sono specificati i dettagli implementativi dei metodi, ma soltanto il modo in cui possono essere invocati;
- un'interfaccia può contenere costanti e firme di metodi, completate con l'indicazione dei tipi restituiti;
- un'interfaccia non può contenere variabili, costruttori o corpo dei metodi;
- un'interfaccia non può essere istanziata, però può avere classi che la implementano;
- una classe può implementare contemporaneamente più interfacce (metodi con uguale firma avranno la stessa implementazione);
- le interfacce possono ereditare da altre interfacce e su di esse è possibile applicare il principio di sostituzione di Liskov (vedi 4.2.1).

Ad esempio:

```
public interface Strumento {
    public void produciSuono();
}
```

[1] http://www.dia.uniroma3.it/~merialdo/didattica/aa2005-2006/poo/trasparenze/POO-07-polimorfismo-interfacce.ppt

Figura 4.1: Esempio di interfaccia e possibili implementazioni

definisce il tipo di oggetti che possono offrire il metodo `produciSuono()`. Tale interfaccia può essere così implementata:

```
public class Tamburo implements Strumento {
  public void produciSuono() {
    System.out.println("bum-bum-bum");
  }
}
```

Una classe che implementa una `interface` può avere altri metodi, oltre a quelli della `interface`, specifici della classe:

```
public class Chitarra implements Strumento {
  private int[] corde;
  public Chitarra(){
    corde = new int[6];
  }
  public void produciSuono() {
    System.out.println("dlen-dlen-dlen");
  }
  public int accorda(int corda, int val) {
        return corde[corda] += val;
  }
}
```

Applicando i concetti dell'ereditarietà, un'interfaccia è la superclasse di tutte le sottoclassi che la implementano (vedi figura 4.1).

4.2 Sostituzione e Polimorfismo

Per un eventuale approfondimento degli argomenti esposti in questa sezione, vedi:

Java e il polimorfismo
a cura della Redazione di IoProgrammo [2]

4.2.1 Principio di sostituzione di Liskov

Il principio di sostituzione di Liskov è una particolare definizione di sottotipo (vedi articolo su Wikipedia [3]). Tale principio si basa sulla nozione di *sostituibilità* secondo cui, *se in un programma*

[2] http://www.itportal.it/developer/java/polimorfismo_34/default.asp
[3] http://it.wikipedia.org/wiki/Principio_di_sostituzione_di_Liskov

S è un sottotipo di T, allora oggetti dichiarati di tipo T possono essere sostituiti con oggetti di tipo S, senza modificare alcuna delle funzionalità richieste al programma (correttezza dei risultati prodotti, compiti svolti, ecc.).

E' quindi possibile, nell'invocazione di un metodo `suona(Strumento s)`, usare come argomento un oggetto istanza di una qualunque classe che implementi l'interfaccia `Strumento`. Ad esempio:

```
public static void main(String[] args){
  Strumento chitarra = new Chitarra();
  Strumento tamburo = new Tamburo();
  Musicista ludovico = new Musicista("Ludovico");
  ludovico.suona(chitarra);
  ludivico.suona(tamburo);
}
```

4.2.2 Polimorfismo

Il *polimorfismo* è considerato, dopo l'incapsulamento e l'ereditarietà, il terzo pilastro della programmazione a oggetti. Questa importantissima tecnica ci aiuta a progettare e scrivere programmi eleganti, solidi e soprattutto espandibili.

Grazie al principio di sostituzione, una funzione può conoscere una "caratteristica generale" di un oggetto, come l'interfaccia della sua superclasse, e lasciare all'oggetto la responsabilità di comportarsi secondo le sue caratteristiche particolari, cioè in base alla particolare implementazione dei metodi della superclasse. *Questa caratteristica del linguaggio, grazie alla quale è possibile usare un oggetto senza sapere esattamente di che tipo sia e come si comporterà, è detta polimorfismo.*

Esempio di applicazione del polimorfismo

```
public class Animale
{
public void interroga()
   {
      System.out.println("Grunt");
   }
}
public class Ghepardo extends Animale
{
public void interroga()
   {
      System.out.println("Groar!");
   }
}
public class Muflone extends Animale
{
public void interroga()
    {
       System.out.println("MOOOO!");
    }
public void salta()
   {
      System.out.println("hop!");
   }
}
public class Armadillo extends Animale {}
public class Upcast
{
public static void main(String[] args)
   {
      Armadillo arm = new Armadillo();
      produciVerso(arm);
      Muflone muf = new Muflone();
      produciVerso(muf);
      Ghepardo ghep = new Ghepardo();
```

```
         produciVerso(ghep);
       }
       private static void produciVerso(Animale anim)  // POLIMORFISMO!
      {
         anim.interroga();
      }
}
```

Il metodo `produciVerso()` accetta come parametro un oggetto di tipo `Animale` o istanze di sottoclassi di `Animale`, che vengono convertite con un cast verso l'alto.

Tale metodo non conosce l'esistenza delle classi `Armadillo`, `Muflone` e `Ghepardo`, ma si limita a sapere di ricevere un oggetto di tipo `Animale`. Grazie al *dynamic binding*, cioè in base al fatto che i metodi richiamati sono quelli appartenenti agli oggetti istanziati e non quelli della superclasse, ciascun animale produce il suo verso, con l'eccezione dell'armadillo, che usa l'implementazione della superclasse. L'output è quindi:

```
Grunt
MOOOO!
Groar!
```

4.2.3 Tipo statico, tipo dinamico, bindind dinamico

Il *tipo statico* è quello usato nella dichiarazione di una variabile ed è determinato a tempo di compilazione, il *tipo dinamico* è quello dell'oggetto realmente istanziato. A tempo di esecuzione viene eseguito il metodo del tipo dinamico, anche perché i metodi definiti nelle interfacce non hanno implementazione se non quella delle classi che le implementano.

Il *binding* è l'associazione di un metodo alla rispettiva classe: in Java il binding viene effettuato durante l'esecuzione (dynamic binding), in modo da garantire che venga invocato il metodo corrispondente all'oggetto effettivamente istanziato.

Si noti che, mentre in Java il binding è sempre dinamico, in C++ è statico, eccetto nel caso di metodi virtuali.

4.2.4 Upcasting e Downcasting

La promozione da un tipo ad un suo supertipo, che avviene quando il riferimento ad un oggetto è dichiarato di un tipo e l'oggetto vero e proprio è invece un sottotipo, viene chiamata *upcasting*. Ad esempio:

```
Strumento chitarra = new Chitarra();
```

Un cast da un tipo ad un suo sottotipo (*downcasting* o semplicemente *cast*) deve essere dichiarato in maniera esplicita e presume che il programmatore abbia la certezza di cosa stia facendo. Come indicato nella guida Thinking in Java [4], «*nel momento in cui viene persa l'informazione di un tipo specifico attraverso un upcast (che equivale ad andare verso l'alto nella gerarchia dell'ereditarietà), ha senso, per recuperare il tipo di informazione, usare un downcast, che significa muoversi verso il basso nella gerarchia dell'ereditarietà*». Questa soluzione risulta utile nel caso in cui si voglia richiamare da un oggetto, dichiarato con il tipo della sua superclasse, metodi che non sono presenti nell'interfaccia della superclasse ma soltanto nell'oggetto stesso. Come evidenziato dai commenti del seguente esempio, un downcast scorretto, anche se non verrà segnalato durante la compilazione, genererà un'eccezione a tempo di esecuzione:

```
class Useful {                          // SUPERCLASSE
  public void f() {}
  public void g() {}
}
```

[4]tradotto da: "Designing with inheritance, downcasting and run-time", tratto da "Thinking in Java" di Bruce Eckel, http://www.codeguru.com/java/tij/tij0083.shtml

```
class MoreUseful extends Useful {      // SOTTOCLASSE
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}
public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {        // ARRAY DI OGGETTI DEL TIPO DELLA SUPERCLASSE
      new Useful(),       // ISTANZA DELLA SUPERCLASSE
      new MoreUseful()    // ISTANZA DELLA SOTTOCLASSE
    };
    x[0].f();      // Metodo presente nell'interfaccia della superclasse -> OK
    x[1].g();      // Metodo presente nell'interfaccia della superclasse -> OK
    // x[1].u();   // ERRORE A TEMPO DI COMPILAZIONE! E' necessario un downcast
    ((MoreUseful)x[1]).u(); // Downcast corretto
    ((MoreUseful)x[0]).u(); // Downcast sbagliato: il compilatore lo accetta!
                            // ERRORE A TEMPO DI ESECUZIONE:
                            // Viene lanciata l'eccezione: ClassCastException
  }
}
```

4.2.5 Downcasting sicuro con l'operatore "istanceof" [5]

Dal momento che in una applicazione Java esistono variabili reference in gran numero, può essere utile determinare a tempo di esecuzione il tipo di oggetto che la variabile sta referenziando. A tal fine, Java supporta l'operatore booleano `instanceof`, la cui sintassi è:

```
A instanceof B
```

dove `A` rappresenta una variabile reference e `B` un tipo referenziabile.

Il tipo rappresentato a run-time dalla variabile reference `A` verrà confrontato con il tipo definito da `B` e l'operatore restituirà `true` o `false`. E' quindi possibile richiamare un downcast se il tipo di un oggetto è effettivamente quello previsto, ad esempio:

```
Veicolo v = new Macchina();
if (v instanceof Macchina)
((Macchina)v).suona();
```

[5]parzialmente tratto dal Capitolo 6 di "Java Mattone dopo Mattone" di Massimiliano Tarquini e Alessandro Ligi, http://www.java-net.it/jmonline/cap6/instanceof.htm

Capitolo 5

Le collezioni

Una collection è un oggetto che ragguppa elementi multipli (anche eterogenei) in una singola entità. Le collections sono usate per immagazzinare, recuperare, trattare dati e trasferire gruppi di dati da un metodo ad un altro.

Tipicamente rappresentano dati che formano gruppi "naturali", come una mano di poker (collection di carte), un cartella di posta (collection di e-mail) o un elenco telefonico (collection di mappe nome-numero).

Per un eventuale approfondimento degli argomenti trattati in questo capitolo, vedi:

The Java Tutorial - Collections
di Joshua Bloch [1]

5.1 Collections Framework

Un *collections framework* (letteralmente: "struttura di collezioni") è un'architettura unificata per la rappresentazione e manipolazione di collezioni. Tutti i frameworks di collezioni sono composti da:

Interfacce: Tipi di dati astratti, analizzati nel precedente capitolo, utilizzati in questo caso per rappresentare collezioni. Le interfacce permettono alle collezioni di essere manipolate indipendentemente dal modo specifico in cui i dati sono rappresentati. Nei linguaggi orientati agli oggetti, le interfacce generalmente formano una gerarchia.

Implementazioni: Implementazioni concrete delle interfacce. In pratica, sono strutture dati riutilizzabili.

Algoritmi: Metodi che eseguono operazioni utili, come la ricerca e l'ordinamento, all'interno di oggetti che implementano le interfacce. Gli algoritmi sono polimorfi, cioè lo stesso metodo può essere usato in molte differenti implementazioni di una certa interfaccia. In pratica, gli algoritmi sono funzionalità riutilizzabili.

Benché l'apprendimento dei *collections frameworks* sia sempre stato abbastanza complesso in altri linguaggi, come nel C++, il loro utilizzo in Java è stato sensibilmente semplificato.

5.1.1 Le interfacce

Le *core collection interfaces* descrivono differenti tipi di collezioni, come riportato in figura 5.1.

Collection Rappresenta un gruppo di oggetti, detti elementi. L'interfaccia Collection è il minimo comun denominatore implementato da tutte le collezioni. Alcuni tipi di collezioni

[1] http://java.sun.com/docs/books/tutorial/collections/index.html

Figura 5.1: Core Collection Interfaces

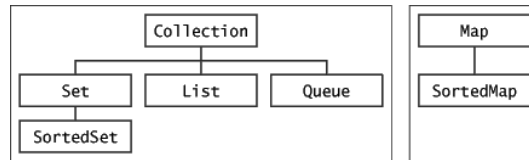



Tabella 5.1: Implementazioni general-purpose

Interfacce	**Implementazioni**			
	Hash table	Resizable Array	Balanced Tree	Linked list
Set	*HashSet*		*TreeSet*	
List		*Array List*		*LinkedList*
Map	*HashMap*		*TreeMap*	

permettono la duplicazione di elementi, altri non la permettono. Alcuni sono ordinati e altri non lo sono. La piattaforma Java non fornisce alcuna implementazione diretta di questa interfaccia ma rende disponibili le implementazioni di sottointerfacce più specifiche, come Set e List.

Set Collezione che non può contenere elementi duplicati. L'interfaccia fornisce un'astrazione del modello matematico di *insieme*.

List Collezione ordinata (detta anche *sequenza*), che può contenere elementi duplicati, ciascuno dei quali è identificato da un indice.

Queue In aggiunta alle operazioni di base sulle collezioni, una Queue (coda) fornisce operazioni aggiuntive e generalmente ordina gli elementi in modalità FIFO (first-in-first-out).

Map Oggetto che collega chiavi a valori. Una mappa non può contenere chiavi duplicate e ogni chiave può essere collegata al massimo con un valore.

SortedSet E' un Set che mantiene gli elementi in ordine crescente. Diverse operazioni aggiuntive sono fornite per usufruire dei vantaggi dell'ordinamento.

SortedMap E' una Map che mantiene il collegamento chiave-valore in ordine crescente di chiave.

5.1.2 Le implementazioni

Le implementazioni più comunemente utilizzate (general-purpose) quelle riportate in tabella 5.1. Il *Java Collections Framework* fornisce diverse implementazioni delle interfacce Set, List e Map. Si noti che le interfacce SortedSet e SortedMap, non riportate nella tabella, hanno le implementazioni *TreeSet* e *TreeMap*, riportate rispettivamente per Set e Map, mentre un'implementazione di Queue è *LinkedList*, che implementa anche List e che fornisce un comportamento FIFO.

Ogni implementazione fornisce tutte le operazioni contenute nella sua interfaccia. Come regola generale, è importante ragionare in termini di interfacce e non di implementazioni, sia perché nella maggior parte dei casi la scelta di un'implementazione ha effetto soltanto sulle prestazioni, sia perché i programmi non dovrebbero dipendere da metodi specifici di particolari implementazioni, in modo da lasciare il programmatore libero di cambiare implementazioni ogni volta che ce ne sia bisogno.

HashSet Gli elementi sono memorizzati in ordine sparso, senza alcuna garanzia sull'ordine in cui potranno essere letti.

TreeSet Gli elementi potranno essere letti in ordine crescente, indipendentemente dall'ordine con cui sono stati inseriti.

ArrayList Implementazione di List come array ridimensionabile.

LinkedList Permette di usare la lista come pila o come coda.

5.2 L'interfaccia Collection

L'interfaccia di Collection è così definita:

```
public interface Collection<E> extends Iterable<E> {
    //Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optional
    boolean remove(Object element); //optional
    Iterator iterator();

    //Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear(); //optional

    //Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

5.2.1 Basic operations

Le operazioni di base sono:

isEmpty Restituisce `true` se la collezione è vuota.

contains Restituisce `true` se l'elemento fa parte della collezione.

add Il metodo è definito in maniera sufficientemente generica da essere applicabile in collezioni che ammettono o non duplicati e garantisce che la collezione conterrà l'elemento specifico dopo il completamento della chiamata. Restituisce `true` se la collezione cambia a seguito della chiamata.

remove Il metodo rimuove una singola istanza dell'elemento specificato dalla collezione, presupponendo che questa lo contenga, e restituisce `true` se la collezione viene modificata.

5.2.2 Iterator

Un iteratore è un oggetto che permette di percorrere una collezione ed eventualmente di rimuoverne elementi. Per avere un Iterator è sufficiente chiamare il corrispondente metodo. L'interfaccia di Iterator è la seguente.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

hasNext Restituisce `true` se c'è un elemento successivo da iterare.

next Restituisce l'elemento successivo della iterazione.

remove Rimuove l'ultimo elemento che è stato restituito da `next()` e può essere chiamato una sola volta per ogni chiamata di `next()`, altrimenti viene invocata una exception.

Esempio di utilizzo di Iterator: un filtro

Si supponga di avere implementato un metodo `cond(Object element)` che restituisca `true` se un elemento deve essere mantenuto nella collezione, `false` altrimenti. Un possibile codice polimorfico sarà il seguente:

```
void filter(Collection x) {
  Iterator i=x.iterator();
     while (i.hasNext()) {
        if (!cond(i.next()))
           i.remove();
     }
}
```

5.2.3 Bulk operations

Le bulk operations (letteralmente "operazioni in blocco") sono particolarmente utili se applicate all'interfaccia Set, perché permettono di applicare su insiemi le operazioni algebriche di unione (addAll), differenza (removeAll) e intersezione (retainAll), oltre a permettere di verificare se un dato insieme è sottoinsieme di un altro (containsAll).

s1.**addAll**(s2) Aggiunge tutti gli elementi di `s2` ad `s1`.

s1.**removeAll**(s2) Rimuove tutti gli elementi di `s1` che sono contenuti anche in `s2`.

s1.**retainAll**(s2) Rimuove da `s1` tutti gli elementi che *non* sono contenuti anche in `s2`, ovvero sono mantenuti gli elementi di `s1` contenuti anche in `s2`.

I metodi `addAll`, `removeAll` e `retainAll` restituiscono `true` se `s1` è stata modificata in seguito all'esecuzione dell'operazione.

s1.**containsAll**(s2) Restituisce `true` se `s1` contiene tutti gli elementi di `s2` .

s1.**clear()** Rimuove tutti gli elementi di `s1`.

Array operations

Le operazioni sugli array sono:

```
Object[] toArray();
Object[] toArray(Object a[]);
```

I metodi `toArray` sono forniti come ponte tra le collezioni e altre API più vecchie, che si aspettano array come input. Le operazioni sugli array permettono di trasformare il contenuto di una collezione in un array.

- La forma semplice, senza argomenti, crea un nuovo array di Object.

- La forma più complessa permette al chiamante di passare un array o di scegliere a tempo di esecuzione il tipo dell'array restituito.

Per riversare il contenuto di `Collection c` in un array di oggetti, la cui lunghezza sarà identica al numero di elementi di `c`, si usa l'istruzione:

```
Object[] a = c.toArray();
```

Se si presume di sapere che `c` contenga solo stringhe, è possibile utilizzare un cast:

```
String[] a = (String[]) c.toArray(new String[0]);
```

5.3 L'interfaccia Set

Un Set, che si basa sul modello matematico di *insieme*, indica una collezione che non può contenere elementi duplicati. Questa interfaccia estende Collection senza aggiungere altri metodi.

Le *Bulk Operations* sono particolarmente utili se applicate a questo tipo di interfaccia, come indicato nella sezione 5.2.3.

5.4 L'interfaccia List

Una List indica una collezione ordinata, detta anche sequenza, nella quale sono aggiunti metodi, rispetto a Collection, per:

- avere un accesso posizionale agli elementi (tramite un indice intero)
- ricercare un elemento e ottenere la sua posizione
- espandere l'Iterator e adattarlo alle caratteristiche posizionali di List
- eseguire operazioni in un determinato intervallo di List.

L'interfaccia di List comprende i seguenti metodi:

```
// Positional Access
Object get(int index);
Object set(int index, Object element);
void add(int index, Object element);
Object remove(int index);
boolean addAll(int index, Collection c);
// Search
int indexOf(Object o);
int lastIndexOf(Object o);
// Iteration
ListIterator listIterator();
ListIterator listIterator(int index);
// Range-view
List subList(int from, int to);
```

5.4.1 La classe Collections

Collections è una classe che contiene metodi di utilità, tra cui:

sort(List) Ordina una lista con un algortimo veloce, che evita di riordinare elementi uguali.

shuffle(List) Scambia gli elementi di una lista in ordine casuale.

reverse(List) Inverte l'ordine degli elementi di una lista.

fill(List, Object) Sovrascrive ogni elemento della lista con l'oggetto specificato.

copy(List dest, List src) Copia la lista sorgente in quella di destinazione.

binarySearch(List, Object) Cerca un elemento in una lista ordinata usando un algoritmo di ricerca binaria.

Nella classe Collections sono anche definite tre costanti, che rappresentano un Set, un List e un Map privi di elementi:

```
Collections.EMPTY_SET
Collections.EMPTY_LIST
Collections.EMPTY_MAP
```

Il principale utilizzo di queste costanti è come input di metodi che una Collection come argomento, ma ai quali non si vuol fornire alcun valore.

5.5 Ordinamento di oggetti

Ci sono due modi per ordinare oggetti:

- l'interfaccia **Comparable** fornisce un "ordine naturale" automatico su quelle classi che lo implementano;
- l'interfaccia **Comparator** dà al programmatore il completo controllo dell'ordinamento.

5.5.1 Interfaccia Comparable

Una lista myList può essere ordinata con il metodo:

```
Collections.sort(myList);
```

Se gli elementi della lista sono stringhe, queste saranno ordinate in ordine alfabetico; se invece sono date, allora verranno posizionate in ordine cronologico. Le classi String e Date, infatti, implementano entrambe l'interfaccia Comparable, fornendo così un "ordine naturale" per l'ordinamento automatico.

Il metodo definito nell'interfaccia di Comparable:

```
int compareTo(Comparable o)
```

compara l'oggetto con quello passato come parametro e restituisce un intero minore, uguale o maggiore di zero se l'oggetto è minore, uguale o maggiore di quello passatogli.

Esempio di implementazione di Comparable

In un piano cartesiano, si vogliono confrontare due punti, sulla base di questa implementazione:

```
Class Point implements Comparable {
int x; int y;
....
int compareTo(Point p) {
// ordino sulle y
retval=y-p.y;
// a partità di y ordino sulle x
if (retval==0) retval=x-p.x;
return retval;
}
```

5.5.2 Interfaccia Comparator

In questo caso si ha un metodo che confronta i due oggetti passatogli come argomenti:

```
int compare(T o1, T o2)
```

Esempio di implementazione di Comparator

Sulla base dell'esempio di implementazione di Comparable, si ha:

```
class NamedPointComparatorByXY
implements Comparator {
int compare (NamedPoint p1, NamedPoint p2) {
// ordino sulle y
retval=p1.y-p2.y;
// a partitÃ  di y ordino sulle x
if (retval==0) retval=p1.x-p2.x;
return retval;
}
```

5.6 Esempio di utilizzo di collezioni

Vedi l'esempio "Tombola" (prof. Marco Ronchetti), disponibile all'indirizzo:
http://latemar.science.unitn.it/marco/Didattica/aa_2005_2006/P2/03_30.pdf

Bibliografia

[1] **Borland.com**
Download di JBuilder
http://www.borland.com/downloads/download_jbuilder.html

[2] **Ciaburro Giuseppe**
Tutorial su Java
http://xoomer.virgilio.it/gciabu/java/tut-java.htm

[3] **De Sio Cesari Claudio**
Object Oriented && Java 5
http://www.claudiodesio.com/download/oo_&&_java_5.zip

[4] **Eclipse open-source community**
http://www.eclipse.org/

[5] **Eckel Bruce**
Thinking in Java
sez. Designing with inheritance, downcasting and run-time
http://www.codeguru.com/java/tij/tij0083.shtml

[6] **Gini Andrea**
Java e UML - Corrispondenza semantica tra Diagrammi di Classe e Codice Java
http://www.lta.disco.unimib.it/didattica/IngSw/slide/Esercitazione-Java-UML.ppt

[7] **IoProgrammo** (redazione di)
Java e il polimorfismo
http://www.itportal.it/developer/java/polimorfismo_34/default.asp

[8] **Merialdo Paolo**
Programmazione Orientata agli Oggetti: Interfacce e Polimorfismo / Upcasting e downcasting
http://www.dia.uniroma3.it/~merialdo/didattica/aa2005-2006/poo/trasparenze/POO-07-polimorfismo-interfacce.ppt

[9] **Mokabyte.it**
Corso Introduttivo su Java
vedi l'indice riportato nella sezione "Testi consigliati" 1.1

[10] **Mulholland Mike**
IM52P Software Design, sez. Information Hiding - Parna's Principles:
http://homepages.north.londonmet.ac.uk/~mulhollm/im52P/week1-2lec.html

[11] **Pighizzini G. e Ferrari M.**
Dai fondamenti agli oggetti - Corso di programmazione Java
ed. Pearson Education Italia, 2005
http://hpe.pearsoned.it/

[12] **Ronchetti Marco**
Corso di Programmazione 2, a.a 2005/2006
http://latemar.science.unitn.it/marco/Didattica/aa_2005_2006/P2/index.html

[13] **Supereva.com**
Introduzione alla OOP (Object-Oriented-Programming)
http://guide.supereva.com/c_/interventi/2000/08/10150.shtml

[14] **WebMasterPoint.org**
Guida a Java
http://www.webmasterpoint.org/java2/

[15] **Wikipedia**
Principio di sostituzione di Liskov
http://it.wikipedia.org/wiki/Principio_di_sostituzione_di_Liskov

[16] **Wikipedia**
David Parnas
http://en.wikipedia.org/wiki/David_Parnas

[17] **WMLScript.it**
Corso Java
http://www.wmlscript.it/java/index.asp

[18] **Sun.com**
The Java Tutorial, a practical guide for programmers
http://java.sun.com/docs/books/tutorial/index.html

[19] **Sun.com**
JavaTM 2 Platform Standard Ed. 5.0 - API Specification
http://java.sun.com/j2se/1.5.0/docs/api/

[20] **Sun.com**
How to Write Doc Comments for the Javadoc Tool
http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

[21] **Tarquini M. e Ligi A.**
Java Mattone dopo Mattone, cap. 6: Ereditarietà
http://www.java-net.it/jmonline/cap6/instanceof.htm